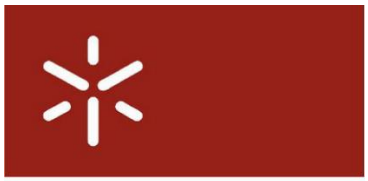


Universidade do Minho

Universidade Do Minho

Escola de Engenharia

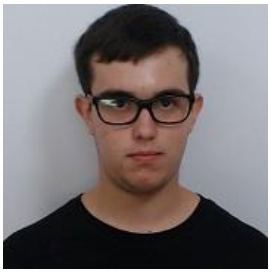
Relatório do Serviço de Controlo e Monitorização de Processos e Comunicação

Sistemas Operativos

LCC 2º Semestre

Junho 2020

Grupo 38



André Araújo

a87987

Henrique Ferreira

a87953

Daniel Ribeiro

a87994

Introdução:

Este trabalho visa a criação de um programa que faça a gestão, controlo e monitorização de diversas tarefas. O programa é constituído por pelo cliente e pelo servidor, o utilizador pode agir no cliente através de 2 modos, por um modo de linguagem textual interpretada (shell) e outra por argumentos.

Depois do utilizador escrever os comandos desejados no cliente, este vai enviar tais comandos ao servidor onde vão ser executados e o seu output colocado no standard output do cliente (caso seja um comando de listagem) ou guardado em um ficheiro log (caso seja o output da execução de uma tarefa).

Funcionalidades implementadas e testes:

Conseguimos implementar todas as funcionalidades pedidas no enunciado e também testar todas elas em diferentes tipos de situações.

Após vários testes, notamos que temos alguns bugs, principalmente na opção adicional, visto que ela ás vezes dá-nos um output maior do que o que devia (concatenando também alguns outputs anterios), e também esta opção ao ser usada pela linha de comandos fecha-nos o servidor, um bug que não conseguimos resolver.

Instruções:

Para executar o programa é nessessário ter o gcc instalado no sistema, para isso abrimos o terminal no linux e digitamos:

```
$ sudo apt install gcc
```

Após a instalação devemos compilar o programa usando o make fornecido, com os comando:

```
$ make all
```

Para a compilação do cliente e do servidor.

Agora com o programa compilado, estamos prontos a trabalhar nele, primeiro devemos ligar o servidor com o comando:

```
$ ./argusd
```

Com ele ligado podemos agora enviar comandos através do cliente, iremos então passar a explicar os diversos comandos disponíveis na aplicação.

Tarefas:

```c
struct Tarefa {
    int numtarefa;
    int inicio;
    int output;
    int *pids;
    int total;
    char * tarefa;
    int comandos;
    int tamoutput;
    int posidx;
    char estado;  //e- execuçao   t - terminada pelo utilizador
                  //   i - terminada por inatividade
                    // m -   terminada por tempo maximo
                    // a - acabou normalmente
                    // k - invalido
};
```

Usamos um array de struct Tarefa para guardar as diversas tarefas da aplicação, juntamente com várias informações relevantes a ela mesmo, como o seu número, o estado, o comando usado para a executar e os pids dos diversos processos para quando tivermos de matar a tarefa.

Comandos

Temos duas formas de executar comandos através do cliente, a primeira na própria linha de comandos do linux usando:

$ ./argus -opção “argumentos”

Onde as opções usadas podem ser: -h, -e, -l, -r, -i, -m, -o e -t

E os argumentos apenas são usados nas opções -e, -i, -m , -o e -t

Ou através duma linha de comandos dentro do cliente usando:

$ ./argus

Irá abrir o cliente e este espera de comandos:

argus$

Agora apenas precisamos de escrever o comando desejado da forma:

argus$ comando “argumentos”

Sendo os comandos disponíveis: ajuda, executar, listar, historico, tempo-inatividade, tempo-execucao e output.

Dos quais apenas o executar, tempo-inatividade, tempo-execucao e output usam argumentos.

Passe-mos então a explicar mais detalhadamente cada uma das opções e como as implementamos em C.

- **Executar uma tarefa**

Opção -e “p1 | p2 . . . | pn” da linha de comandos ou opção executar “p1 | p2 . . . | pn” através da shell.

Por exemplo:

```
$ ./argus -e “ls -l | wc -l”    ou
argus$ executar “ls -l | wc -l”
```

Esta opção vai executar uma dada tarefa e registar o seu output em um ficheiro chamado “Log.txt”.

Esta opção opção começa por dividir os diversos processos num array de arrays, após uma divisão inicial, iremos executar os comandos de forma a que o comando n leia as informações do processo anterior, e escreva para o pipe seguinde daonde o próximo processo vai ler e assim sucessivamento, até que o último processo vai imprimir os dados no ficheiro Log.txt.

Mas entre cada tarefa temos uma espécie de “sensor” encadeado que vai ler do processo anterior e escrever para o seguinde sendo este útil para a opção do tempo de inativade pois este sensor vai ter um alarme no tempo de inatividade e se ele não ler nada durante esse tempo, vai dar return a um valor (neste caso 10) que o pai vai apanhar e sabemos que a tarefa foi terminada por inatividade, este alarme é reinstalado sempre que algo é lido do pipe.

- **Ajuda**

Opção -h da linha de comandos ou opção ajuda através da shell.

```
$ ./argus -h
Argus$ ajuda
```

Esta opção irá imprimir no standard output do client uma lista com os possíveis comandos da aplicação.

- **Terminar uma tarefa**

Opção -t da linha de comandos ou opção terminar através da shell.

```
$ ./argus -t n
Argus$ terminar n
```

Esta opção irá terminar uma tarefa cujo número dela é o n, a implementação dela foi com base numa pesquisa da tarefa no array das tarefas, comparando o número dela, quando encontramos essa tarefa, iremos mandar um SIGKILL a todos os pids dela mesmo, mudamos o seu estado para t e fazemos free dos apontadores.

- **Listar tarefas em execução**

Opção -l da linha de comandos ou opção listar através da shell.

```
$ ./argus -l
Argus$ listar
```

Esta opção vai listar todas as tarefas em execução no standard output do client, para sabermos quais tarefas ainda não terminaram, nós possuimos um array de struct tarefas, essa struct tarefa tem vários atributos de uma dada tarefa sendo um delos o estado da mesma, o estado é um char e se for 'e' quer dizer que está em execução no dado momento.

- **Histórico de tarefas executadas**

Opção -r da linha de comandos ou opção historico através da shell.

```
$ ./argus -r
Argus$ historico
```

Esta opção vai listar o histórico de todas as tarefas já executadas nesta aplicação, assim como a forma com que acabaram, podendo acabar de forma normal, por inatividade dum pipe, por tempo máximo de execução ou a pedido do utilizador com a opção -t n (terminar n).

Quando uma tarefa é terminada a maneira com que foi terminada é guardada na struct dessa mesma tarefa, após isso para esta opção apenas temos de ver todas as tarefas no array de struct tarefas, imprimindo o seu número de tarefa e o seu estado de saída.

- **Tempo de inatividade**

Opção -i da linha de comandos ou opção tempo-inatividade através da shell.

```
$ ./argus -i n
Argus$ tempo-inactividade n
```

Esta opção vai definir o tempo máximo de inatividade de um qualquer pipe anónimo de uma qualquer tarefa. Para matarmos uma tarefa após este tempo de inatividade, incluimos uma espécie de "sensor" entre cada duas tarefas este sensor tem um alarm do tempo de inatividade. Ele vai ler do processo anterior e passar os dados para a seguinte, quando lê dados este alarme é reinstalado, assim sendo se não ler nada durante o tempo de inativade definido, um SIGALARM vai ser ativado e a tarefa irá dar exit com return de 10, este return será apanhado pelo SIG_CHILD handler do pai e sabemos que foi terminada por inatividade, assim o pai faz o resto do tratamento, mandando SIGKILL para todos os pids, colocando o estado a i (terminado por inactividade) e liberta a memória alocada.

- **Tempo máximo de execução**

Opção -m da linha de comandos ou opção tempo-maximo através da shell.

```
$ ./argus -m n
Argus$ tempo-maximo n
```

Esta opção vai definir o tempo máximo de execução de uma dada tarefa sendo isto o tempo-máximo que uma tarefa pode estar a executar a contar desde o último exec. Para matarmos uma tarefa após o tempo máximo é implementado um alarm(1) a contar desde o último exec e sempre que o SIGALRM é acionado, iremos colocar outro alarm(1), incrementamos os segundos de todas as tarefas em execução (com estado e) e iremos ver se alguma destas têm os segundos maior ou igual ao tempo máximo de execução, se tiver iremos matar a tarefa, enviando um SIGKILL a todos os pids, colocando o estado como m (terminada por tempo máximo) e damos free á memória alocada.

Funcionalidade adicional:

(Possui alguns bugs, ver parte dos testes)

É nos dada a informação de que devemos guardar o standard output produzido por cada tarefa num ficheiro "log", e que devemos utilizar um outro ficheiro "log.idx" que guarda a posição no ficheiro "log" de onde devemos ler o output.

Como as tarefas nem sempre acabam de executar sequencialmente , surge a necessidade de guardar a posição de cada tarefa no

ficheiro "log.idx" , sendo assim adicionada a variável posidx em cada struct Tarefa , também achamos melhor que cada tarefa

guarda-se o tamanho do seu output, pois ia facilitar muito a leitura de dados, caso contrário teríamos de andar a procurar no

ficheiro "log.idx" até onde é que deviamos ler. Logo cada struct Tarefa terá também uma variável tamoutput.

Mas como descobrimos esse tal tamanho do standart output de cada tarefa?

Primeiro vamos ver quando e como é que nós guardamos esta informação no "log".

Imaginemos a tarefa cmd1 | cmd2 | ... | cmdn-1 | cmdn , o output será no guardado no ficheiro devido a um redirecionamento do std out para o file descriptor do "log" { dup2 (log_fd , 1) }, na linha de código anterior ao exec do cmdn .

Como o exec vai substituir o binário do programa original, não vamos conseguir descobrir esse tamanho de output aqui. Mas sabemos que quando tratamos o SIGCHLD

do processo filho que executou o cmdn , a tarefa ja acabou e o standard output dessa tarefa já foi adicionado ao ficheiro "log". Logo será no sigchld_handler onde

vamos descobrir quantos bytes é foram escritos no ficheiro.

Para utilizar o valor de variáveis no handler, elas têm de ser globais, logo criámos duas variáveis globais , a primeira chamada "poslog" que equivale ao número de bytes já

escritos no ficheiro log, sendo fácil de perceber que o primeiro byte do std out da próxima tarefa vai ficar na posição poslog + 1. A segunda chamada "posidx" que equivale ao

número de bytes já escritos no ficheiro "log.idx".

Sequencialmente o processo que ocorre para cada tarefa i , com i ∈ [1,∞[:

Quando o server é executado começamos com : poslog = 0, posidx = 0

1 -> int file_fd = ("log" , O_RDONLY);

2 -> int tamLogAposTarefa = lseek (file_fd , 0 , SEEK_END); // posicionar o file_fd no fim do ficheiro ficando assim a saber quantos bytes estão no ficheiro após a execução da tarefa i

3 -> tarefas[i].tamOutput = tamLogAposTarefa - poslog; // novo tamanho do ficheiro - o tamanho do ficheiro antes da execução da tarefa a ser tratada (tarefa i);

4 -> int fileidx_fd = ("log.idx" , O_RDONLY);

5 -> tarefas[i].posidx = posidx + 1; // como "posidx" equivale ao número de bytes já escritos no ficheiro "log.idx", a posiçao no log onde se encontra o std out da tarefa i vai ser escrita apartir de posidx + 1

6 -> sprintf (bufAux , "%d" , poslog + 1); // escrevemos num buf auxiliar a posiçao no log onde se vai encontar o stdout da tarefa i

7 -> strcat (bufAux , " "); // adicionamos um espaço entre cada posição escrita no log.idx para facilitar procura

8 -> write (fileidx_fd , bufAux , tamanhobufAux);

9 -> posLog = tamLogAposTarefa; // novo tamanho do log é o tamanho após a execução da tarefa

10 -> posidx += tamanhobufAux; // adicionamos tamanhobufAux bytes pois foi o número de bytes escritos em log.idx

Agora quando o cliente executa a opção -o n , o processo é simples:

1 -> int fileidx_fd = ("log.idx" , O_RDONLY);

2 -> lseek (fileidx_fd , tarefas[tarefan].posidx , SEEK_SET); // colocamo-nos na posicao posidx da tarefa n no log.idx

3 -> read (fileidx_fd , aux , 10); //lemos para um buf auxiliar

4 -> int posLog = atoi(aux); // retiramos o inteiro correspondete a posição do stdout no ficheiro log

5 -> file_fd = open ("log",O_RDONLY);

6 -> lseek (file_fd , posLog , SEEK_SET);

7 -> lidos = read (file_fd , buff , tarefas[ntarefa].tamoutput); // lemos para o buff o tamoutput da tarefa pretendida, e escrevemos para o fifo.

Conclusão:

Achamos este projeto de grande importância visto que abordou praticamente todos os conceitos abordados ao longo deste semestre onde tivemos de aplicar-los numa aplicação de maior escala.

Os nossos objetivos foram alcançados visto que conseguimos cumprir todos os requesitos básicos e o opcional da aplicação desta forma estamos contentes com o resultado final, mesmo assim achamos que algumas das nossas soluções podiam ser melhores, como por exemplo no tempo de execução máximo, podia-mos ter colocado um alarme (tempo-execução) no último processo da aplicação e também lhe dar um handler respetivo que iria dar um valor de saída diferente caso saísse por tempo máximo de execução, e o pai depois apanhava o valor de saída e colocava o estado respetivo.